# Implementing a Microcontroller Watchdog with a Field-Programmable Gate Array (FPGA)

Bartholomew F. Straka

John F. Kennedy Space Center

Major: Electrical Engineering

USRP Spring 2013 Session

Date: 12 APR 13

# Implementing a Microcontroller Watchdog with a Field-Programmable Gate Array (FPGA)

Bartholomew F. Straka[1]
*University of Central Florida, Orlando, Florida, 32816*

**Reliability is crucial to safety. Redundancy of important system components greatly enhances reliability and hence safety. Field-Programmable Gate Arrays (FPGAs) are useful for monitoring systems and handling the logic necessary to keep them running with minimal interruption when individual components fail. A complete microcontroller watchdog with logic for failure handling can be implemented in a hardware description language (HDL). HDL-based designs are vendor-independent and can be used on many FPGAs with low overhead.**

## Nomenclature

| | | |
|---|---|---|
| *ADC* | = | Analog-to-Digital Converter |
| *DAC* | = | Digital-to-Analog Converter |
| *DMR* | = | Dual Modular Redundant |
| *FPGA* | = | Field-Programmable Gate Array |
| *HDL* | = | Hardware Description Language |
| *IDE* | = | Integrated Development Environment |
| *IEEE* | = | Institute of Electrical and Electronics Engineers |
| *I/O* | = | Input/Output |
| *IP Core* | = | Intellectual Property Core. A functional black box unit provided for use in an FPGA design as licensed intellectual property. |
| *K-Map* | = | Karnaugh Map |
| *LAS* | = | Launch Abort System |
| *LFSR* | = | Linear Feedback Shift Register |
| *VHDL* | = | VHSIC (Very High Speed Integrated Circuits) Hardware Description Language |
| *POR* | = | Power-On Reset |
| *WDT* | = | Watchdog Timer |

## I. Introduction

Safety is the cornerstone of the National Aeronautics and Space Administration's (NASA) core values.[1] Extraordinary accomplishments often come with extraordinary risks. One way to minimize risk and account for the unexpected is through system redundancy. By duplicating (or even triplicating) the most critical elements of a system, overall reliability is enhanced. The trade-off for greater reliability through redundancy is additional overhead in terms of total cost, weight, system complexity, or other factors. For matters involving safety, however, reliability should not be compromised to the greatest extent possible.

The basic implementation of redundancy is a primary and alternate pair. The alternate takes control when the primary fails. Redundancy can also exist in systems such as a bridge with many suspension cables, where the failure of individual cables steadily degrades the reliability of the bridge. In the bridge example, redundancy is inherent. With a primary and alternate pair, often some form of active monitoring must be present to detect the failure of the primary and bring the alternate online. This process for monitoring and switching to ensure system reliability is known as voting logic.[2]

Implementing a system watchdog with a Field-Programmable Gate Array (FPGA) is an example of voting logic. FPGAs allow custom logic circuits to be designed and programmed as de facto hardware. An FPGA's logic can be simulated quickly, processes can be performed in parallel, and can be easily reconfigured if problems are found or

[1] Intern, Flight Instrumentation Branch, John F. Kennedy Space Center, University of Central Florida.

updated later with improvements. The possible failure of the voting logic itself, however implemented, is an additional consideration. For example, sensitivity to radiation at the gate level is a concern.[2] Therefore, selection of appropriately radiation-tolerant devices may be critical to the reliability of such a voting logic application.

The specific application discussed herein is for a Dual Modular Redundant (DMR) microcontroller pair. In this DMR system, the two microcontrollers are fed the same inputs and operate in parallel. The output of one processor is actively controlling the end-item devices (valves, thrust vector controllers, solid state switches, etc.) while the output of the other (standby) controller is inhibited. When the primary fails, the standby resumes the desired functions with minimal interruption. Each microcontroller generates a heartbeat signal that is monitored by an FPGA. A disruption in the heartbeat of the initially-active microcontroller will cause the FPGA to switch control to the alternate microcontroller.

Although the system considered is DMR, the techniques can be extended to several levels of redundancy. If the probability of failure for each identical element is considered independent, then the probability of failure for the system decreases dramatically. Mathematically, the probability of two independent events $A$ and $B$ both happening is

$$P(A \text{ and } B) = P(A \cap B) = P(A)P(B) \tag{1}$$

If the probability of such failures were exactly equal for the identical elements $A$, then the probability of total failure $F$ decreases exponentially with the number of redundant layers $x$, since P($A$) is hopefully less than one.

$$P(F) = P(A)^x \tag{2}$$

These conditions are ideal, however, and never realizable. Additionally, the conditions that lead to the failure of one element may influence the failure of another identical element. So if the probability of failure is reasonably low to begin with, only a few layers of redundancy are needed to greatly reduce the probability of total failure. Accordingly, the overhead associated with too much redundancy has diminishing return if the elements have the same vulnerabilities and will be subjected to the same stresses.

## II. Watchdog Realization

A watchdog timer (WDT) is a timer of fixed or specified duration that must be renewed by the system being watched to avoid timing out. If the WDT expires, it is a secondary indication of some problem with the system under observation. Many modern microcontrollers and other embedded systems come with a WDT already integrated. The typical corrective action when the WDT expires is to reset the microcontroller. The timer durations are often limited to a few fixed durations. While this may be a sensible solution for remote sensors that periodically measure something mundane, the results could be disastrous for some critical component such as the processor controlling thrust vectoring for a Launch Abort System (LAS).

For critical applications, monitoring the system with an external watchdog has several advantages. The external monitor may be designed to a certain set of specifications, such that it is not limited to the settings of the on-board WDT. The external monitor may be able to take corrective action sooner than the WDT would to reset. An external monitor will detect a complete failure of an element where the integrated WDT also fails. Most importantly, an external monitor can multiplex a functioning element to the output if an active element fails, thereby minimizing interruption in system operation.

An FPGA is well-suited to perform the external monitoring and voting logic, since it is highly customizable, can perform a wide variety of tasks in parallel (perhaps in addition to the voting logic), and often offer a large number of input/output (I/O) pins. The features make an FPGA a robust and modular addition with an acceptably small footprint for many system designs. It may well be the case that an FPGA is already being used in a design for some other purpose, and the voting logic functionality can be added as a convenience.

The caveat is that FPGAs are usually digital devices only, meaning that it is best suited for cases where the heartbeat or other parameter of a system being monitored is a digital signal. There are mixed-signal FPGAs capable of analog-to-digital conversion (ADC), digital-to-analog conversion (DAC), and other functions commonly integrated with microcontrollers, while some FPGAs themselves incorporate a complete microcontroller. Such devices may allow for a combination of traditional serial programming in a language like C and parallel programming in a hardware description language (HDL).The implementation considered here is not for a mixed-signal device and assumes a digital heartbeat signal from the system being observed.

### A. Watchdog Structure

One consideration in designing a watchdog is what constitutes a failure based upon the heartbeat signal received. The simplest watchdog will renew its timer every time a pulse (also called a "kick") is received as long as the timer has not expired. That may mean that a heartbeat signal which suddenly increases in frequency, one that slows and just barely pulses before the watchdog expires, or just remains at a high logic level will all satisfy the watchdog. This simple watchdog is just a window of time, where any pulse that does not exceed the allowed time limit is acceptable. This may work for many applications, but a much stricter watchdog is achievable.

One benefit of considering more than just the rising pulse of a heartbeat is faster detection of potential failure. By considering the heartbeat signal in more detail, an inappropriate rising edge may be an earlier indicator of failure that allows a faster changeover to a still-functioning microcontroller. There are a few other indicators to check, and since the FPGA can operate these checks in parallel, there is not much overhead to using them other than a relatively small amount of FPGA gates.

If the microcontroller outputs a digital signal heartbeat with a known duty cycle of 50% (the watchdog may be adjusted for any duty cycle, but 50% is considered here), the watchdog can use the expected rise and fall times to detect a failure sometimes faster than one clock cycle of the heartbeat. The watchdog detailed here will test for the following conditions: signal is stuck low does not rise fast enough, signal is stuck high or does not fall fast enough, signal rises outside an acceptable window of time, or signal is completely lost (or tri-stated). Signal may be lost due to a dramatic occurrence such as the device suddenly catching fire or the connection may be weak. Either way, the FPGA will control the multiplexing of which microcontroller is active, so a lost signal is always a failure for this consideration.

**B. Counters**

The building block for realizing the aforementioned watchdog structure is a counter. For maximum flexibility and utility, the counter is described in Very High Speed Integrated Circuits Hardware Description Language (VHDL). Each FPGA vendor provides its own Integrated Development Environment (IDE). These IDEs usually include a method to design the logic circuits on the FPGA using schematic capture, which is a visual representation of logic gates and boxes. A basic block such as a counter may have different characteristics from one IDE to another. VHDL can be easily imported and modified for use with any FPGA. The counters described here will also include a few more features than are commonly provided with the schematic capture blocks that make them more adaptable to different heartbeat signals.

A basic counter takes in a clock signal and increments or decrements an internal register with each specified clock event, either rising or falling edge. The counter may incorporate a reset signal input that resets the count to the original value and a terminal count output that is asserted if the counter reaches its final value without being reset.

The first type of counter used in the watchdog is intended to create an acceptable window of time for a rising edge of the heartbeat signal. This is accomplished with an active-low output. The counter is reset with each rising heartbeat and is expected to reach terminal count before the next rising edge occurs, which should be during the finite terminal count window.

A D flip-flop (seen as *Rising_Edge_Trigger* in Fig. 3) is employed to detect a bad rising edge. The terminal count of the counter is fed to the D input of the flip-flop, and the heartbeat signal is fed to the Clk input. A rising edge of the heartbeat outputs the D input to the Q output of the flip-flop. If the rising edge occurs during an active-low terminal count, the output is low and there is no error. If the rising edge occurs outside the terminal count window, the output is high and there is an error. Note that although the rising edge resets the counter (and the terminal count goes high), the input of the flip-flop should be low at the time of the rising edge. The events happen in parallel, and there is always some propagation delay in FPGA's before gate states can make a logical transition. Accordingly, the output of the counter will go high when reset after the flip-flop acts on its original input. Figure 1[3] illustrates the action of the acceptable window counter and watchdog in simulation.


**Figure 1. Counter for generating acceptable rising edge window.** *The first two heartbeats reset the counter while terminal count (D) is asserted low. The third rising edge is late, the full terminal count window is visible, and the error is detected. The image is stretched for improved visibility of the waveforms.*

The second type of counter is a simple timeout, where the terminal count itself is the error indicator. If the counter is not reset in time, the watchdog is not satisfied. To detect errors faster, two counters are employed as timeouts. One is reset by a rising edge and the other by a falling edge. Note that it is also possible to implement the edge window counter for a falling edge instead of a rising edge, or both for the potential to detect an error even faster. Figure 2[3] illustrates the action of the timeout counter and watchdog in simulation for a rising edge reset. The action is the opposite for a falling edge reset.


**Figure 2. Counter that times out without a rising edge.** *The first two heartbeats reset the counter normally. The third rising edge is late and the error is detected. The image is stretched for improved visibility of the waveforms.*

A schematic visualization of the entire watchdog is pictured in Fig. 3[4]. The actual counters and watchdog are written in VHDL, but the schematic capture here provides a block diagram view.

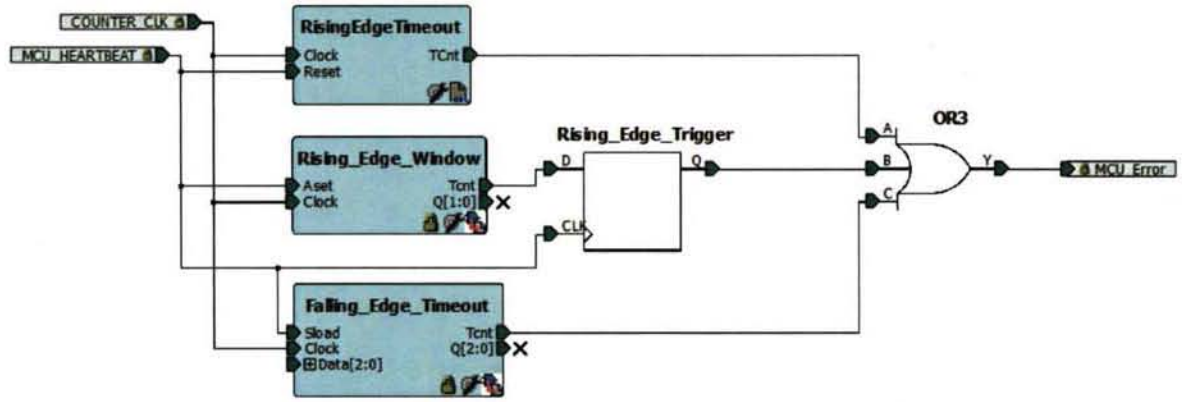


**Figure 3. Schematic capture view of the watchdog with counters.** *A microcontroller failure is inferred from an error indicator on any of the three counters watching the signal.*

Using VHDL offers some advantages over the schematic capture. A major benefit is customization through the use of "generics." Generics are used as numbers in a VHDL entity and have a default constant value, but a new value may be passed in when an entity is instantiated in a higher-level design. This can allow for a single entity, such as a counter, to be used in many different configurations. For instance, notice that the counter in Fig. 2 increments the counter on a rising edge and has a maximum count of five. The counter is reset at the very last possible moment (the simulation is pre-synthesis and includes no timing), so there is a possibility that when timing is involved this will trigger many false errors. So for instance, the count may easily be increased to six to account for this possibility.

An example implementation of this highly configurable counter is given below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity CustomCounter is

generic (
--! Adjust the duration of counter here. Default is for a system clock 10x the pulse signal.
Count                   : integer := 5;                 --! Count

--! Controls whether the edge timeout is for a high or low signal.
--! '0' will create a falling edge timeout (times out if pulse stays high)
--! '1' will create a rising edge timeout (times out if pulse stays low) [default]
```

```
EdgeType            : std_logic := '1';          --! Specifies type of timeout

--! Determines whether the counter acts on the falling [default] or rising edge of its driving clock.
--! '0' selects the counter to act on the clock's falling edge
--! '1' selects the counter to act on the clock's rising edge
CounterDrive        : std_logic := '1';          --! Specifies driving edge for counter clock

--! Determines whether output is active high or active low. Active low is used to create a
--! window for acceptable clock events.
--! '0' will be active high [default]
--! '1' will be active low
OutputHL            : std_logic := '1';          --! Specifies active High/Low output

--! Adjust the duration (number of clock cycles) for which the counter's terminal count signal
--! is asserted. Default is one clock cycle; increase for the rising edge window to have more
--! tolerance and account for a pulse signal not synchronized with the system clock.
--! Value cannot exceed "Count" since it is reset at the end of the count.
TCntDur             : integer := 0               --! Duration of TCnt in clock cycles
);

port (
--! Inputs
Clock               : in std_logic;              --! Clock
Reset               : in std_logic;              --! Reset

--! Outputs
TCnt                : out std_logic              --! Terminal Count (Active High)
);

end CustomCounter;

architecture Behaviour of CustomCounter is

--! The following function is a modification of the default rising_edge function provided in the
--! Institute of Electrical and Electronics Engineers (IEEE) standard library.
--! This function checks for a legitimate edge transition by checking the state previous to the clock event.
--! Something like clock'Event and clock = '1', while commonly used, does not account for the fact
--! That the previous state may be other logic states, such as U or Z
FUNCTION action_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN IS
BEGIN
RETURN (s'EVENT AND (To_X01(s) = CounterDrive) AND
(To_X01(s'LAST_VALUE) = not CounterDrive));
END;

-- Signal Declarations
signal InternalCount    : integer;               --! Maintains internal count
signal AssertTime       : integer := TCntDur;    --! Maintains TCnt Duration
signal ResetCount       : std_logic;             --! Indicates terminal count
signal TcntInternal     : std_logic := OutputHL; --! Initializes output to desired value

begin
ResetCount <= '1' when InternalCount = Count else '0';

-- Process(es)
--! Counting process that resets the internal count on asynchronous reset and loops after reaching final count.
Counting : process (Reset, Clock)
begin
if (Reset = EdgeType) then
    InternalCount <= 0;
elsif action_edge(Clock) then
    if ResetCount = '1' then
        InternalCount <= 0;
```

```
    else
        InternalCount <= InternalCount + 1;
    end if;
end if;
end process;

--! Will assert the terminal count high for specified number of clock cycles.
Enabling : process (Reset, Clock)
begin
if (Reset = EdgeType)then
    TCntInternal <= OutputHL;
    AssertTime <= TCntDur;
elsif action_edge(Clock) then
    if ResetCount = '1' then
        TCntInternal <= not OutputHL;
        AssertTime <= 0;
    else
        if AssertTime = TCntDur then
            TCntInternal <= OutputHL;
        else
            AssertTime <= AssertTime + 1;
            TCntInternal <= not OutputHL;
        end if;
    end if;
end if;
end process;

TCnt <= TCntInternal;

-- End Architecture
end Behaviour;
```

To put these counters to use as a watchdog, as visualized in Fig. 3, they must be instantiated and some of the generics set. Below is the VHDL watchdog using the custom counter.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

--! @details
--! The watchdog incorporates both a rising edge and falling edge timeout as well as
--! an acceptable window of tolerance for a rising edge. The +- tolerance of this window
--! is adjustable here by altering the counters or by changing the clock frequency.

entity Watchdog is

port (
--! Inputs
HeartBeat           : in STD_LOGIC;          --! Active-high reset.
Clock               : in STD_LOGIC;          --! Clock.
Initialize          : in STD_LOGIC;          --! Initialize to erase start-up error

--! Outputs
--! Each output has a different meaning. In this project, the 'Error' output will remain
--! asserted forever if a failure event has ever occurred. The 'Status' indicator is asserted
--! (asserted here is a logic '1' by default) whenever any kind of failure is detected, but is
--! not permanently asserted like the 'Error' indicator. If the pulse signal eventually "feeds
--! the watchdog" back to a state of normalcy, the 'Status' output will remain unasserted.
--! The remaining outputs indicate the particular type of failure, mostly for debugging and
--! possibly for detailed failure analysis.
Error               : out STD_LOGIC;         --! Indicates if a failure ever occurred.
```

```
Status          : out STD_LOGIC;        --! Present state of failure.
StuckHigh       : out STD_LOGIC;        --! Pulse is stuck high.
StuckLow        : out STD_LOGIC;        --! Pulse is stuck low.
BadRise         : out STD_LOGIC         --! Rising edge outside acceptable window.
);

end Watchdog;

architecture Behaviour of Watchdog is

--! Instantiations of sub-components.
component CustomCounter
generic (
--! These are the same as in the CustomCounter code, but must be adjusted in the mapping, not here.
Count           : integer := 5;         --! Count
EdgeType        : std_logic := '1';     --! Specifies type of timeout
CounterDrive    : std_logic := '1';     --! Specifies driving edge for counter clock
OutputHL        : std_logic := '1';     --! Specifies active High/Low output
TCntDur         : integer := 0          --! Duration of TCnt in clock cycles
);

--! Ports
port (
--! Inputs
Clock : in std_logic; --! Clock
Reset : in std_logic; --! Reset

--! Outputs
TCnt : out std_logic --! Terminal Count
);
end component;

--! Signal Declarations
--! Initialize to some known value when/if possible to avoid unknown logic states.
signal D              : std_logic := '0';
signal Q              : std_logic := '0';
signal NoFallingEdge  : std_logic := '0';
signal NoRisingEdge   : std_logic := '0';
signal BadRisingEdge  : std_logic := '0';
signal StatusSig      : std_logic := '0';

--! Begin Behaviour
Begin

--! @Instantiations
FallingEdgeTimeout : CustomCounter
--! @details Times out without falling edge after specified interval.
--! Detects a pulse signal stuck high.
Generic MAP(
Count => 5,                             --! Times out after 5 clock cycles
EdgeType => '0',                        --! Falling edge resets counter
CounterDrive => '1',                    --! Counts on rising edge
OutputHL => '0',                        --! Active-high output
TCntDur => 5                            --! Timeout asserted for 1 clock cycle
)
-- port map
port map(
-- Inputs
Clock => Clock,
Reset => HeartBeat,
TCnt => NoFallingEdge
);
```

```
RisingEdgeTimeout : CustomCounter
--! @details Times out without rising edge after specified interval.
--! Detects a pulse signal stuck low.
Generic MAP(
Count => 5,                                  --! Times out after 5 clock cycles
EdgeType => '1',                             --! Rising edge resets counter
CounterDrive => '1',                         --! Counts on rising edge
OutputHL => '0',                             --! Active-high output
TCntDur => 5                                 --! Timeout asserted for 1 clock cycle
)
-- port map
port map(
-- Inputs
Clock => Clock,
Reset => HeartBeat,
TCnt => NoRisingEdge
);

RisingEdgeWindow : CustomCounter
--! @details Creates a window for an acceptable rising edge clock event.
--! Detects a sudden incorrect pulse rise.
Generic MAP(
Count => 4,                                  --! Times out after 4 clock cycles
EdgeType => '1',                             --! Rising edge resets counter
CounterDrive => '0',                         --! Counts on falling edge
OutputHL => '1',                             --! Active-low output
TCntDur => 0                                 --! Timeout asserted for 1 clock cycle
)
-- port map
port map(
-- Inputs
Clock => Clock,
Reset => HeartBeat,
TCnt => D
);
--! @End Instantiations

--! @Processes
DFF : process (HeartBeat)
--! @details Implements a D flip-flop to ensure the rising edge of the microcontroller pulse is within the
--! window established by the RisingEdgeWindow instantiation. The active-low output of the RisingEdgeWindow
--! is fed to the data input of the D flip-flop, while the pulse signal is treated as the flip-flop's clock
--! signal. The rising edge of the pulse will cause the data input to appear on Q, the flip-flop's output.
--! Since the window created by RisingEdgeWindow is active-low, any rising edge pulse signal that occurs
--! outside this duration of time will cause the normally-high output of RisingEdgeWindow to appear on Q
--! and indicate an error.
--! Note that to reduce false triggers rising_edge(signal) is used instead of (signal'event and signal = '1').
--! The difference is that rising_edge() ensures the previous logic state was '0', whereas clk'event detects
--! a change from any state, including Z, U, X, W, L, H.
begin
if rising_edge(HeartBeat) then               --! Check for "Enable" of Flip-Flop
    BadRisingEdge <= D;
end if;                                      --! Q unchanged without enable and clock
end process;

DFF2 : process (StatusSig)
--! @details Latches onto errors permanently, or ignores them if being initialized.
begin
if (Initialize = '1') then
    Q <= '0';
elsif (StatusSig = '1') then                 --! Check for "Enable" of Flip-Flop
```

```
    Q <= '1';
end if;                                          --! Q unchanged without enable and clock
end process;
--! @End Processes

--! @details Error signal is triggered from any of the error sources.
StuckHigh <= NoFallingEdge;
StuckLow <= NoRisingEdge;
BadRise <= BadRisingEdge;
StatusSig <= NoRisingEdge or NoFallingEdge or BadRisingEdge;
Status <= StatusSig;
Error <= Q;
--! End of Architecture Body
end Behaviour;
```

## III. Voting Logic

In addition to a watchdog for detecting errors, a layer of voting logic is necessary for a DMR system to function. The first step is to monitor both microcontrollers with the watchdog. Once an error is detected, the specifications and designer decide what to do with that information. Standard practice with a simple digital design may be to use a Karnaugh Map (K-Map) for finding a simplified realization. With just a few added I/O options the K-map can become tedious, especially if the system has greater complexity than dual redundancy. The underlying voting logic, which is to select a healthy active component, is very straightforward when requirements are kept to a minimum. A simple schematic capture of an early version of the DMR voting logic used here is pictured in Fig. 4[4].

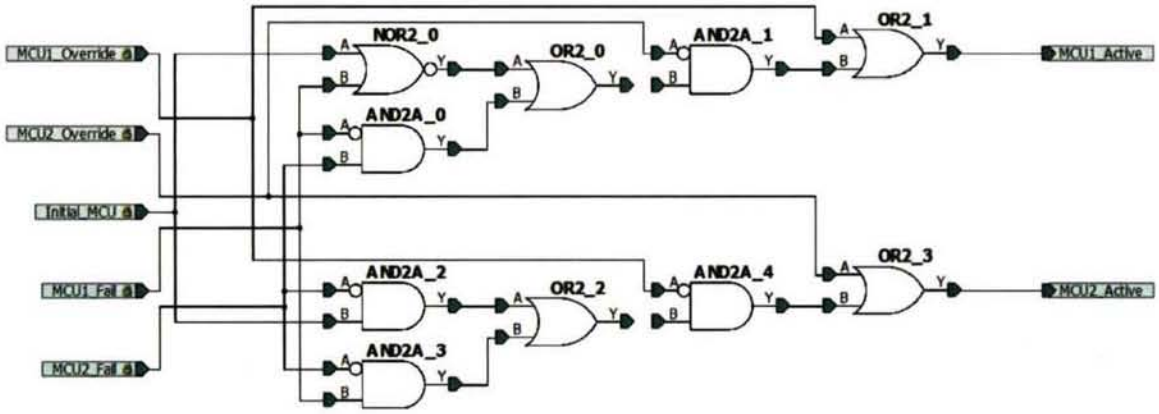


**Figure 4. Schematic capture view of simple voting logic.** *The voting logic here also accounts for an initial microcontroller selection and an override.*

VHDL again can provide a relatively simpler, more easily modified (without studying a mass of logic gates each time) solution that could transfer readily to FPGAs from different vendors. Some example code is given below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

--! @details
--! Switches from an initially-selected processor to an alternate one when a failure
--! of the initial processor is indicated. Overrides are possible for extreme cases
--! where both processors fail.

entity ProcessorSelect is

generic (
--! Choose the initially active processor.
```

```
--! '0' is Processor 1 [default]
--! '1' is Processor 2
InitProcessor : std_logic := '0' --! Initial Processor Select
);

port (
-- Inputs
--! Indicates whether the failure status of a processor has ever been triggered.
MCU1Fail            : in std_logic;             --! MCU1 Failure Indicator
MCU2Fail            : in std_logic;             --! MCU2 Failure Indicator

--! The status indicator monitors the original failure trigger. The MCU1Fail and MCU2Fail
--! are permanently latched from these signals. Ordinarily, once a processor has an indicated
--! failure it should never be active again. In the event that both processors fail, however,
--! the original status trigger can be monitored to select the most viable alternative.
MCU1Status          : in std_logic;             --! MCU1 current status
MCU2Status          : in std_logic;             --! MCU2 current status

--! Enables manual override, must be disabled by default!
--! '0' is disable
--! '1' is enable
ForceEnable         : in std_logic;             --! Override MCU Selection

--! Processor to select once manual override enabled. Should not be unknown!
--! '0' is Processor 1
--! '1' is Processor 2
ForceSelect         : in std_logic;             --! Override select for MCU2

-- Outputs
--! The output selects and indicates the appropriate processor. Only one at a time may ever be active.
--! '0' is Processor 1.
--! '1' is Processor 2.
SelectedProcessor   : out std_logic             --! Selects appropriate processor.
);

end ProcessorSelect;

architecture Behaviour of ProcessorSelect is

-- Signal Declarations
signal ActiveProcessor : std_logic := InitProcessor;    --! Active Processor

begin
-- Process(es)
--! Selects alternate processor in the event of failure; decides what to do when both fail.
Selection : process (ForceEnable)
begin
    if (ForceEnable = '1') then                         --! Manual override
        ActiveProcessor <= ForceSelect;                 --! Forced processor selection
    else                                                --! No override; normal operation
        if (MCU1Fail = '1') then
            ActiveProcessor <= '1'; --! Choose Processor 2 when 1 fails
        elsif (MCU2Fail = '1') then
            ActiveProcessor <= '0'; --! Choose Processor 1 when 2 fails
        elsif (MCU1Fail = '1' and MCU2Fail = '1') then      --! Both processors have failed
            if (MCU1Status = '1' and MCU2Status = '0') then     --! Processor 2 watchdog currently good
                ActiveProcessor <= '1'; --! Select processor 2
            elsif (MCU1Status = '0' and MCU2Status = '1') then  --! Processor 1 watchdog currently good
                ActiveProcessor <= '0'; --! Select processor 1
            else
                ActiveProcessor <= InitProcessor;           --! Ensure a processor is always selected
            end if;
```

```
        else
            ActiveProcessor <= InitProcessor;          --! No failures; continue to use initial.
        end if;
end if;
end process;

SelectedProcessor <= ActiveProcessor;

-- End Architecture
end Behaviour;
```

Many other voting logic algorithms are possible, depending on the application. Writing the logic in VHDL will help simplify the revision process if needed. For instance, if both processors fail a linear feedback shift register (LFSR) might be used to pseudo-randomly choose one rather than the initial default. That design may be more useful for applications requiring more layers of redundancy. Figure 5[4] shows the overall DMR microcontroller handling (watchdog plus voting logic) block diagram.

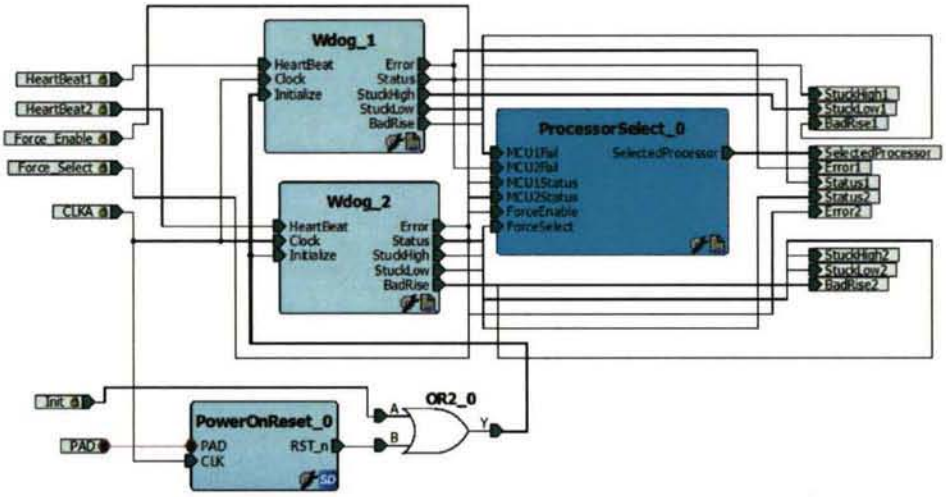


**Figure 5. Overall block diagram of the watchdog with voting logic.** *The watchdogs monitor the microcontrollers and the voting logic tries to keep the system functioning with minimal interruption in the event of a failure. Note the Power-On Reset (POR) circuit. This initializes the system to a known state when first powered and is vendor-specific.*

## IV. Considerations

The POR circuit seen in Fig. 5 is necessary to ensure the system starts up properly and without any false failures while the system clock and heartbeat signals are stabilized. The one pictured is vendor-specific to Microsemi/Actel FPGAs and was found in an application note[5]. One benefit of the circuit is the ability to modify how long the initialization lasts. Similar PORs may be available for other vendors, and it is worth consideration to ensure that all logic starts at a known state.

The schematic capture and VHDL code found herein is provided for example only. Actual implementation is specific to a particular application and must take the appropriate requirements into account. The intent was to convey a simple watchdog capable of monitoring both clock edges for identifying a variety of possible failures very quickly.

Capabilities of FPGAs may vary, and each vendor offers its own set of Intellectual Property Cores (IP Cores) for use in designing. These cores may be extremely useful but are not open source and available to use on other FPGAs. This may limit the reuse of a particular design, no matter how much was written in standard VHDL, without access to comparable IP Cores from other vendors.

## V. Conclusion

Redundancy can provide a dramatic increase in reliability. Reliability is not to be compromised where safety or high cost are concerned. Voting logic and watchdogs are necessary for many redundant systems. The use of FPGAs allows for this functionality to be added with a low footprint or readily added to designs incorporating FPGAs. Using VHDL for FPGA design allows for more modular use of the design, as it is not vendor-specific. Maintaining well-managed and well-commented VHDL designs will allow for reuse of code in many future designs

## Acknowledgments

I extend my gratitude to the Undergraduate Student Research Program (USRP) for providing funding that made this internship opportunity available. I also thank my mentor, Peter T. Johnson, for directing me to this interesting project.

## References

[1]National Aeronautics and Space Administration, "Governance and Strategic Management Handbook," *NASA Policy Directive (NPD) 1000.0A*, 1 August 2008, p. 3.

[2]Thompson, S., and Mycroft, A., "Abstract Interpretation in Space: SET Immunity of Majority Voting Logic," *Proc. APPSEM II Workshop,* Frauenchiemsee, Germany, 2005, pp. 5-8.

[3]ModelSim ACTEL, Software Package, Ver. 10.1b, Mentor Graphics Corporation, Wilsonville, OR, 2012.

[4]Libero, Software Package, Ver. 10.1.3.1, Microsemi Corporation, Aliso Viejo, CA, 2013.

[5]Microsemi, "Internal Power-on reset and Post Programming Reset Circuit for Flash-Based FPGAs," *Application Note AC380*, 2012.